2.4 **The Structure of Agents**

So far we have talked about agents by describing behavior—the action that is performed after any given sequence of percepts. Now we must bite the bullet and talk about how the insides work. The job of AI is to design an agent program that implements the agent function—the mapping from percepts to actions. We assume this program will run on some sort of computing device with physical sensors and actuators—we call this the agent architecture

Agent = architecture + program.

Agent program

Agent architecture

Obviously, the program we choose has to be one that is appropriate for the architecture. If the program is going to recommend actions like Walk, the architecture had better have legs. The architecture might be just an ordinary PC, or it might be a robotic car with several onboard computers, cameras, and other sensors. In general, the architecture makes the percepts from the sensors available to the program, runs the program, and feeds the program's action choices to the actuators as they are generated. Most of this book is about designing agent programs, although Chapters 25 and 26 deal directly with the sensors and actuators.

2.4.1 Agent programs

The agent programs that we design in this book all have the same skeleton: they take the current percept as input from the sensors and return an action to the actuators. Notice the difference between the agent program, which takes the current percept as input, and the agent function, which may depend on the entire percept history. The agent program has no agent = architecture + program . □□ 5choice but to take just the current percept as input because nothing more is available from the environment; if the agent's actions need to depend on the entire percept sequence, the agent will have to remember the percepts.

5 There are other choices for the agent program skeleton; for example, we could have the agent programs be coroutines that run asynchronously with the environment. Each such coroutine has an input and output port and consists of a loop that reads the input port for percepts and writes actions to the output port.

We describe the agent programs in the simple pseudocode language that is defined in Appendix B . (The online code repository contains implementations in real programming languages.) For example, Figure 2.7 shows a rather trivial agent program that keeps track of the percept sequence and then uses it to index into a table of actions to decide what to do. The table—an example of which is given for the vacuum world in Figure 2.3 — represents explicitly the agent function that the agent program

embodies. To build a rational agent in this way, we as designers must construct a table that contains the appropriate action for every possible percept sequence.

Figure 2.7

```
function TABLE-DRIVEN-AGENT(percept) returns an action
  persistent: percepts, a sequence, initially empty
              table, a table of actions, indexed by percept sequences, initially fully specified

  append percept to the end of percepts
  action ← LOOKUP(percepts, table)
  return action
```

Figure 2.7 The TABLE-DRIVEN-AGENT program is invoked for each new percept and returns an action each time. It retains the complete percept sequence in memory.

It is instructive to consider why the table-driven approach to agent construction is doomed to failure. Let be the set of possible percepts and let be the lifetime of the agent (the total number of percepts it will receive). The lookup table will contain entries. Consider the automated taxi: the visual input from a single camera (eight cameras is typical) comes in at the rate of roughly 70 megabytes per second (30 frames per second, pixels with 24 bits of color information). This gives a lookup table with over entries for an hour's driving. Even the lookup table for chess—a tiny, well-behaved fragment of the real world—has (it turns out) at least entries. In comparison, the number of atoms in the observable universe is less than . The daunting size of these tables means that (a) no physical agent in this universe will have the space to store the table; (b) the □ □ □ P T ∑T t = 1 | |P| | t 1080 × 720 10 600,000,000,000 10 150 10 80designer would not have time to create the table; and (c) no agent could ever learn all the right table entries from its experience.

Despite all this, TABLE-DRIVEN-AGENT does do what we want, assuming the table is filled in correctly: it implements the desired agent function.

The key challenge for AI is to find out how to write programs that, to the extent possible, produce rational behavior from a smallish program rather than from a vast table.

We have many examples showing that this can be done successfully in other areas: for example, the huge tables of square roots used by engineers and schoolchildren prior to the 1970s have now been replaced by a five-line program for Newton's method running on electronic calculators. The question is, can AI do for general intelligent behavior what Newton did for square roots? We believe the answer is yes.

In the remainder of this section, we outline four basic kinds of agent programs that embody the principles underlying almost all intelligent systems:

Simple reflex agents;

Model-based reflex agents;

Goal-based agents; and

Utility-based agents.

Each kind of agent program combines particular components in particular ways to generate actions. Section 2.4.6 explains in general terms how to convert all these agents into learning agents that can improve the performance of their components so as to generate better actions. Finally, Section 2.4.7 describes the variety of ways in which the components themselves can be represented within the agent. This variety provides a major organizing principle for the field and for the book itself.

2.4.2 Simple reflex agents

The simplest kind of agent is the simple reflex agent. These agents select actions on the basis of the current percept, ignoring the rest of the percept history. For example, the vacuum agent whose agent function is tabulated in Figure 2.3 is a simple reflex agent, because its decision is based only on the current location and on whether that location contains dirt. An agent program for this agent is shown in Figure 2.8 .

Figure 2.8

```
function REFLEX-VACUUM-AGENT([location,status]) returns an action
    if status = Dirty then return Suck
    else if location = A then return Right
    else if location = B then return Left
```

The agent program for a simple reflex agent in the two-location vacuum environment. This program implements the agent function tabulated in Figure 2.3.

Figure 2.8 The agent program for a simple reflex agent in the two-location vacuum environment. This program implements the agent function tabulated in Figure 2.3 .

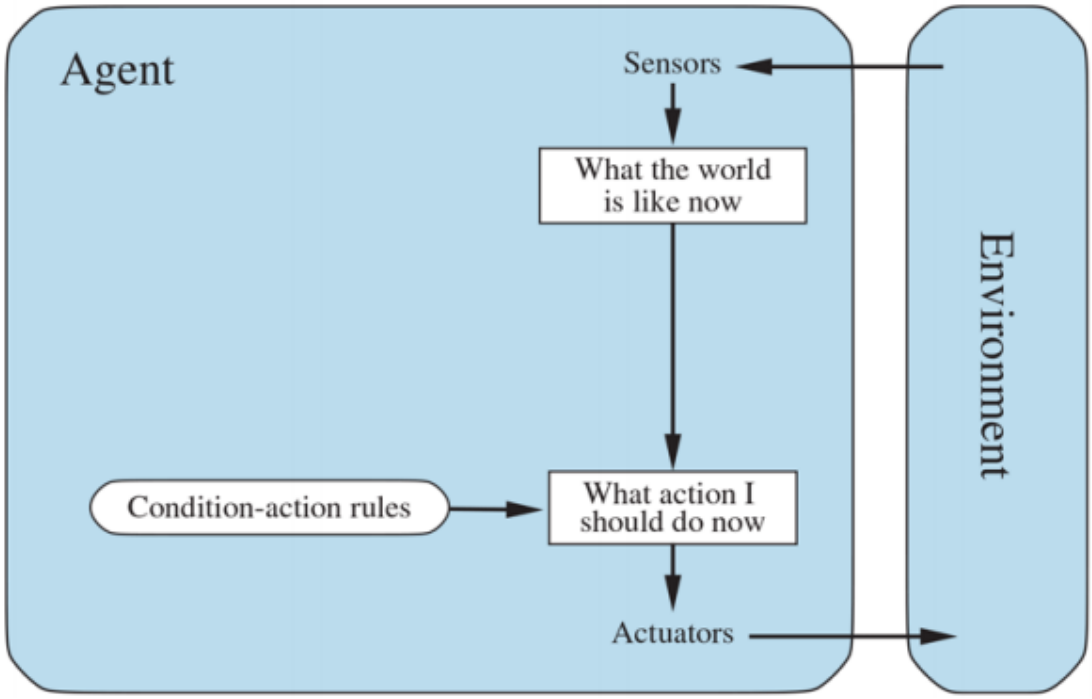
Simple reflex agent Notice that the vacuum agent program is very small indeed compared to the corresponding table. The most obvious reduction comes from ignoring the percept history, which cuts down the number of relevant percept sequences from to just 4. A further, small reduction comes from the fact that when the current square is dirty, the action does not depend on the location. Although we have written the agent program using if-then-else statements, it is simple enough that it can also be implemented as a Boolean circuit. Simple reflex behaviors occur even in more complex environments. Imagine yourself as the driver of the automated taxi. If the car in front brakes and its brake lights come on, then you should notice this and initiate braking. In other words, some processing is done on the visual input to establish the condition we call “The car in front is braking.” Then, this triggers some established connection in the agent program to the action “initiate braking.” We call such a connection a condition–action rule, written as

6 Also called situation–action rules, productions, or if–then rules. if car-in-front-is-braking then initiate-braking.

□□ 4 T 6condition–action rule Humans also have many such connections, some of which are learned responses (as for driving) and some of which are innate reflexes (such as blinking when something approaches the eye). In the course of the book, we show several different ways in which such connections can be learned and implemented.

The program in Figure 2.8 is specific to one particular vacuum environment. A more general and flexible approach is first to build a general-purpose interpreter for condition– action rules and then to create rule sets for specific task environments. Figure 2.9 gives the structure of this general program in schematic form, showing how the condition–action rules allow the agent to make the connection from percept to action. Do not worry if this seems trivial; it gets more interesting shortly.

Figure 2.9



Schematic diagram of a simple reflex agent. We use rectangles to denote the current internal state of the agent's decision process, and ovals to represent the background information used in the process.

Figure 2.9 Schematic diagram of a simple reflex agent. We use rectangles to denote the current internal state of the agent's decision process, and ovals to represent the background information used in the process.

An agent program for Figure 2.9 is shown in Figure 2.10 . The INTERPRET-INPUT function generates an abstracted description of the current state from the percept, and the RULE MATCH function returns the first rule in the set of rules that matches the given state description. Note that the description in terms of "rules" and "matching" is purely conceptual; as noted above, actual implementations can be as simple as a collection of logic gates implementing a Boolean circuit. Alternatively, a "neural" circuit can be used, where the logic gates are replaced by the nonlinear units of artificial neural networks (see Chapter 21).

Figure 2.10

```
function SIMPLE-REFLEX-AGENT(percept) returns an action
  persistent: rules, a set of condition–action rules

  state ← INTERPRET-INPUT(percept)
  rule ← RULE-MATCH(state, rules)
  action ← rule.ACTION
  return action
```

A simple reflex agent. It acts according to a rule whose condition matches the current state, as defined by the percept.

Figure 2.10 A simple reflex agent. It acts according to a rule whose condition matches the current state, as defined by the percept.

Simple reflex agents have the admirable property of being simple, but they are of limited intelligence. The agent in Figure 2.10 will work only if the correct decision can be made on the basis of just the current percept—that is, only if the environment is fully observable.

Even a little bit of unobservability can cause serious trouble. For example, the braking rule given earlier assumes that the condition car-in-front-is-braking can be determined from the current percept—a single frame of video. This works if the car in front has a centrally mounted (and hence uniquely identifiable) brake light. Unfortunately, older models have different configurations of taillights, brake lights, and turn-signal lights, and it is not always possible to tell from a single image whether the car is braking or simply has its taillights on. A simple reflex agent driving behind such a car would either brake continuously and unnecessarily, or, worse, never brake at all.

We can see a similar problem arising in the vacuum world. Suppose that a simple reflex vacuum agent is deprived of its location sensor and has only a dirt sensor. Such an agent has just two possible percepts: and . It can in response to ; what □□□□ [Dirty] [Clean] Suck [Dirty]should it do in response to ? Moving fails (forever) if it happens to start in square , and moving fails (forever) if it happens to start in square . Infinite loops are often unavoidable for simple reflex agents operating in partially observable environments.

Escape from infinite loops is possible if the agent can randomize its actions. For example, if the vacuum agent perceives , it might flip a coin to choose between and . It is easy to show that the agent will reach the other square in an average of two steps. Then, if that square is dirty, the agent will clean it and the task will be complete. Hence, a randomized simple reflex agent might outperform a deterministic simple reflex agent.

Randomization

We mentioned in Section 2.3 that randomized behavior of the right kind can be rational in some multiagent environments. In single-agent environments, randomization is usually not rational. It is a useful trick that helps a simple reflex agent in some situations, but in most cases we can do much better with more sophisticated deterministic agents.

2.4.3 Model-based reflex agents The most effective way to handle partial observability is for the agent to keep track of the part of the world it can't see now. That is, the agent should maintain some sort of internal state that depends on the percept history and thereby reflects at least some of the unobserved aspects of the current state. For the braking problem, the internal state is not too extensive—just the previous frame from the camera, allowing the agent to detect when two red lights at the edge of the vehicle go on or off simultaneously. For other driving tasks such as changing lanes, the agent needs to keep track of where the other cars are if it can't see them all at once. And for any driving to be possible at all, the agent needs to keep track of where its keys are.

Internal state

Updating this internal state information as time goes by requires two kinds of knowledge to be encoded in the agent program in some form. First, we need some information about how the world changes over time, which can be divided roughly into two parts: the effects of the agent's actions and how the world evolves independently of the agent. For example, when the agent turns the steering wheel clockwise, the car turns to the right, and when it's raining the car's cameras can get wet. This knowledge about "how the world works"—whether implemented in simple Boolean circuits or in complete scientific theories—is called a transition model of the world.

Transition model

Second, we need some information about how the state of the world is reflected in the agent's percepts. For example, when the car in front initiates braking, one or more illuminated red regions appear in the forward-facing camera image, and, when the camera gets wet, droplet-shaped objects appear in the image partially obscuring the road. This kind of knowledge is called a sensor model.

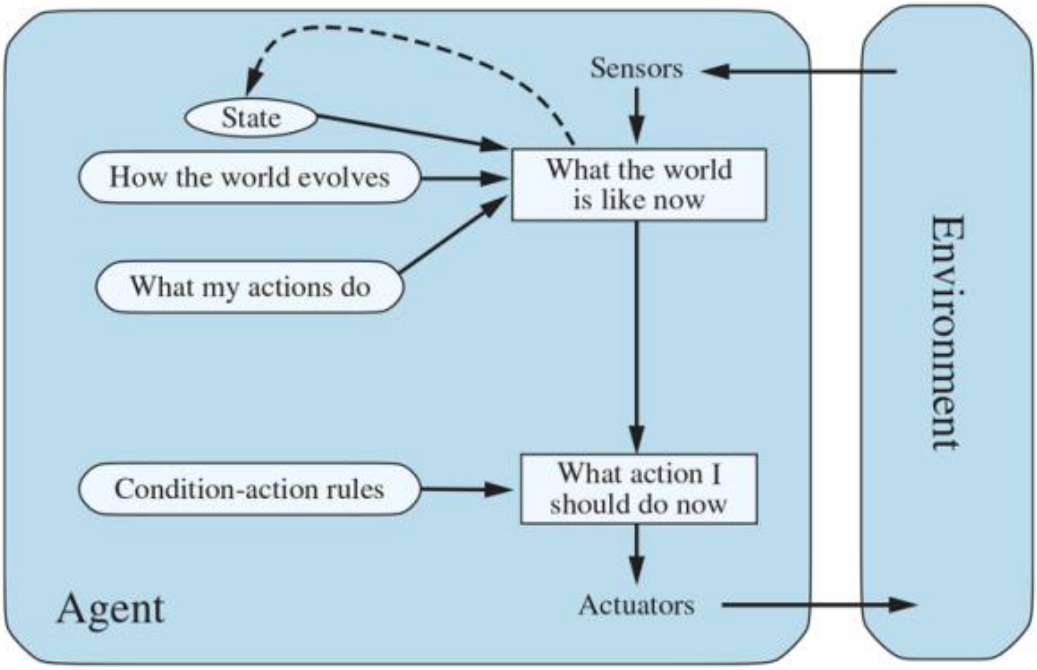
Sensor model

Together, the transition model and sensor model allow an agent to keep track of the state of the world—to the extent possible given the limitations of the agent's sensors. An agent that uses such models is called a model-based agent.

Model-based agent

Figure 2.11 gives the structure of the model-based reflex agent with internal state, showing how the current percept is combined with the old internal state to generate the updated description of the current state, based on the agent's model of how the world works. The agent program is shown in Figure 2.12 . The interesting part is the function UPDATE-STATE, which is responsible for creating the new internal state description. The details of how models and states are represented vary widely depending on the type of environment and the particular technology used in the agent design.

Figure 2.11



A model-based reflex agent.

Figure 2.12

Figure 2.11 A model-based reflex agent.

```
function MODEL-BASED-REFLEX-AGENT(percept) returns an action
  persistent: state, the agent's current conception of the world state
              transition_model, a description of how the next state depends on
                    the current state and action
              sensor_model, a description of how the current world state is reflected
                    in the agent's percepts
              rules, a set of condition–action rules
              action, the most recent action, initially none

  state ← UPDATE-STATE(state, action, percept, transition_model, sensor_model)
  rule ← RULE-MATCH(state, rules)
  action ← rule.ACTION
  return action
```

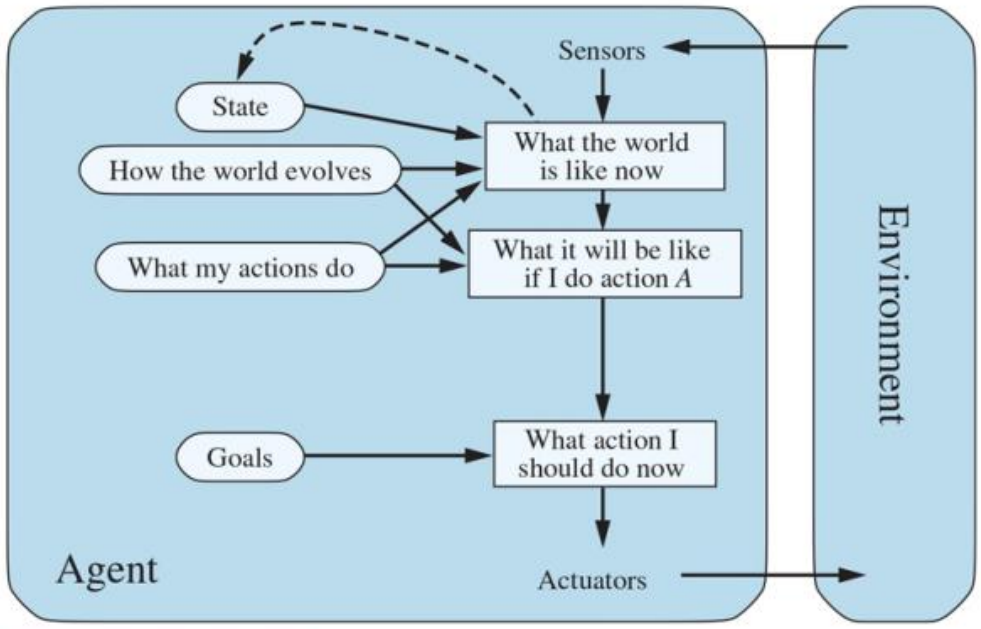
A model-based reflex agent. It keeps track of the current state of the world, using an internal model. It then chooses an action in the same way as the reflex agent.

Figure 2.12 A model-based reflex agent. It keeps track of the current state of the world, using an internal model. It then chooses an action in the same way as the reflex agent.

Regardless of the kind of representation used, it is seldom possible for the agent to determine the current state of a partially observable environment exactly. Instead, the box labeled “what the world is like now” (Figure 2.11) represents the agent’s “best guess” (or sometimes best guesses, if the agent entertains multiple possibilities). For example, an automated taxi may not be able to see around the large truck that has stopped in front of it and can only guess about what may be causing the hold-up. Thus, uncertainty about the current state may be unavoidable, but the agent still has to make a decision.

2.4.4 Goal-based agents

Knowing something about the current state of the environment is not always enough to decide what to do. For example, at a road junction, the taxi can turn left, turn right, or go straight on. The correct decision depends on where the taxi is trying to get to. In other words, as well as a current state description, the agent needs some sort of goal information that describes situations that are desirable—for example, being at a particular destination. The agent program can combine this with the model (the same information as was used in the model-based reflex agent) to choose actions that achieve the goal. Figure 2.13 shows the goal-based agent’s structure.



A model-based, goal-based agent. It keeps track of the world state as well as a set of goals it is trying to achieve, and chooses an action that will (eventually) lead to the achievement of its goals.

Goal

Figure 2.13 A model-based, goal-based agent. It keeps track of the world state as well as a set of goals it is trying to achieve, and chooses an action that will (eventually) lead to the achievement of its goals.

Goal

Sometimes goal-based action selection is straightforward—for example, when goal satisfaction results immediately from a single action. Sometimes it will be more tricky—for example, when the agent has to consider long sequences of twists and turns in order to find a way to achieve the goal. Search (Chapters 3 to 5) and planning (Chapter 11) are the subfields of AI devoted to finding action sequences that achieve the agent's goals.

Notice that decision making of this kind is fundamentally different from the condition– action rules described earlier, in that it involves consideration of the future—both "What will happen if I do such-and-such?" and "Will that make me happy?" In the reflex agent designs, this information is not explicitly represented, because the built-in rules map directly from percepts to actions. The reflex agent brakes when it sees brake lights, period. It has no idea why. A goal-based agent brakes when it sees brake lights because that's the only action that it predicts will achieve its goal of not hitting other cars.

Although the goal-based agent appears less efficient, it is more flexible because the knowledge that supports its decisions is represented explicitly and can be modified. For example, a goal-based agent's behavior can easily be changed to go to a different destination, simply by specifying that destination as the goal. The reflex agent's rules for when to turn and when to go straight will work only for a single destination; they must all be replaced to go somewhere new.

2.4.5 Utility-based agents

Goals alone are not enough to generate high-quality behavior in most environments. For example, many action sequences will get the taxi to its destination (thereby achieving the goal), but some are quicker, safer, more reliable, or cheaper than others. Goals just provide a crude binary distinction between "happy" and "unhappy" states. A more general performance measure should allow a comparison of different world states according to exactly how happy they would make the agent. Because "happy" does not sound very scientific, economists and computer scientists use the term utility instead.

7 The word "utility" here refers to "the quality of being useful," not to the electric company or waterworks.

Utility We have already seen that a performance measure assigns a score to any given sequence of environment states, so it can easily distinguish between more and less desirable ways of getting to the taxi's destination. An agent's utility function is essentially an internalization of the performance measure. Provided that the internal utility function and the external performance measure are in agreement, an agent that chooses actions to maximize its utility will be rational according to the external performance measure.

Utility function

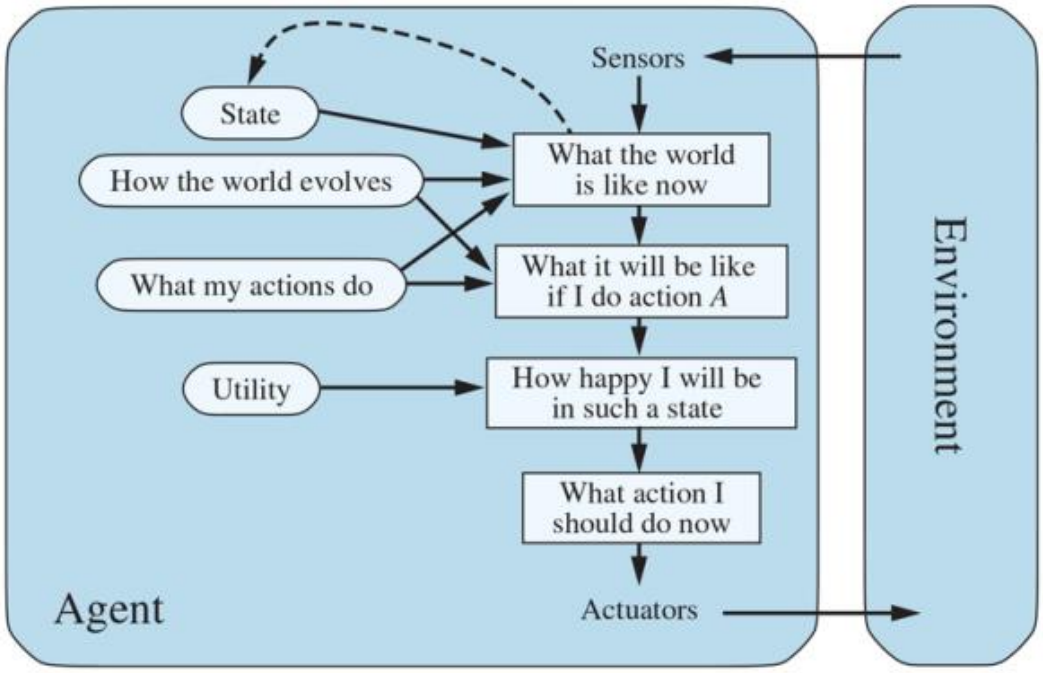
7Let us emphasize again that this is not the only way to be rational—we have already seen a rational agent program for the vacuum world (Figure 2.8) that has no idea what its utility function is—but, like goal-based agents, a utility-based agent has many advantages in terms of flexibility and learning. Furthermore, in two kinds of cases, goals are inadequate but a utility-based agent can still make rational decisions. First, when there are conflicting goals, only some of which can be achieved (for example, speed and safety), the utility function specifies the appropriate tradeoff. Second, when there are several goals that the agent can aim for, none of which can be achieved with certainty, utility provides a way in which the likelihood of success can be weighed against the importance of the goals.

Partial observability and nondeterminism are ubiquitous in the real world, and so, therefore, is decision making under uncertainty. Technically speaking, a rational utility-based agent chooses the action that maximizes the expected utility of the action outcomes—that is, the utility the agent expects to derive, on average, given the probabilities and utilities of each outcome. (Appendix A defines expectation more precisely.) In Chapter 16 , we show that any rational agent must behave as if it possesses a utility function whose expected value it tries to maximize. An agent that possesses an explicit utility function can make rational decisions with a general-purpose algorithm that does not depend on the specific utility function being maximized. In this way, the "global" definition of rationality—designating as rational those agent functions that have the highest performance—is turned into a "local" constraint on rational-agent designs that can be expressed in a simple program.

Expected utility

The utility-based agent structure appears in Figure 2.14 . Utility-based agent programs appear in Chapters 16 and 17 , where we design decision-making agents that must handle the uncertainty inherent in nondeterministic or partially observable environments. Decision making in multiagent environments is also studied in the framework of utility theory, as explained in Chapter 18 .

Figure 2.14



A model-based, utility-based agent. It uses a model of the world, along with a utility function that measures its preferences among states of the world. Then it chooses the action that leads to the best expected utility, where expected utility is computed by averaging over all possible outcome states, weighted by the probability of the outcome.

Figure 2.14 A model-based, utility-based agent. It uses a model of the world, along with a utility function that measures its preferences among states of the world. Then it chooses the action that leads to the best expected utility, where expected utility is computed by averaging over all possible outcome states, weighted by the probability of the outcome.

At this point, the reader may be wondering, “Is it that simple? We just build agents that maximize expected utility, and we’re done?” It’s true that such agents would be intelligent, but it’s not simple. A utility-based agent has to model and keep track of its environment, tasks that have involved a great deal of research on perception, representation, reasoning, and learning. The results of this research fill many of the chapters of this book. Choosing the utility-maximizing course of action is also a difficult task, requiring ingenious algorithms that fill several more chapters. Even with these algorithms, perfect rationality is usually unachievable in practice because of computational complexity, as we noted in Chapter 1 . We also note that not all utility-based agents are model-based; we will see in Chapters 22 and 26 that a model-free agent can learn what action is best in a particular situation without ever learning exactly how that action changes the environment.

Model-free agent

Finally, all of this assumes that the designer can specify the utility function correctly; Chapters 17 , 18 , and 22 consider the issue of unknown utility functions in more depth.

2.4.6 Learning agents

We have described agent programs with various methods for selecting actions. We have not, so far, explained how the agent programs come into being. In his famous early paper, Turing (1950) considers the idea of actually programming his intelligent machines by hand. He estimates how much work this might take and concludes, “Some more expeditious method seems desirable.” The method he proposes is to build learning machines and then to teach them. In many areas of AI, this is now the preferred method for creating state-of-the-art systems. Any type of agent (model-based, goal-based, utility-based, etc.) can be built as a learning agent (or not).

Learning has another advantage, as we noted earlier: it allows the agent to operate in initially unknown environments and to become more competent than its initial knowledge alone might allow. In this section, we briefly introduce the main ideas of learning agents. Throughout the book, we comment on opportunities and methods for learning in particular kinds of agents. Chapters 19 – 22 go into much more depth on the learning algorithms themselves.

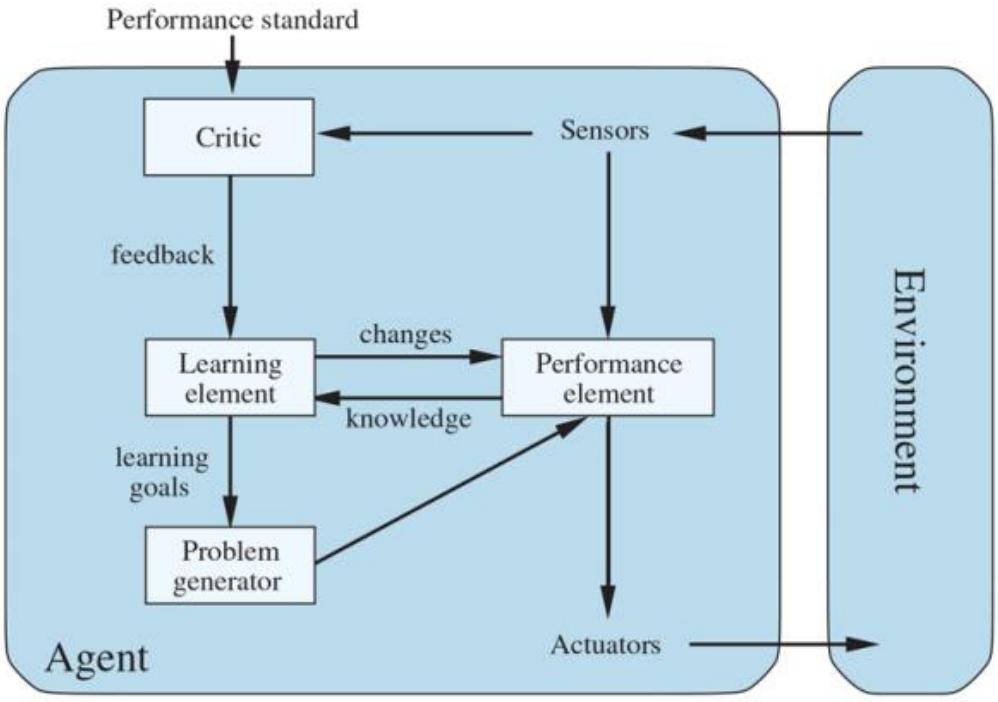
Learning element

Performance element

A learning agent can be divided into four conceptual components, as shown in Figure 2.15 . The most important distinction is between the learning element, which is responsible for making improvements, and the performance element, which is responsible for selecting external actions. The performance element is what we have previously considered to be the entire agent: it takes in percepts and decides on actions. The learning element uses feedback from the critic on how the agent is doing and determines how the performance element should be modified to do better in the future.

Figure 2.15

Figure 2.15



A general learning agent. The "performance element" box represents what we have previously considered to be the whole agent program. Now, the "learning element" box gets to modify that program to improve its performance.

A general learning agent. The “performance element” box represents what we have previously considered to be the whole agent program. Now, the “learning element” box gets to modify that program to improve its performance.

Critic

The design of the learning element depends very much on the design of the performance element. When trying to design an agent that learns a certain capability, the first question is not “How am I going to get it to learn this?” but “What kind of performance element will myagent use to do this once it has learned how?” Given a design for the performance element, learning mechanisms can be constructed to improve every part of the agent.

The critic tells the learning element how well the agent is doing with respect to a fixed performance standard. The critic is necessary because the percepts themselves provide no indication of the agent’s success. For example, a chess program could receive a percept indicating that it has checkmated its opponent, but it needs a performance standard to know that this is a good thing; the percept itself does not say so. It is important that the performance standard be fixed. Conceptually, one should think of it as being outside the agent altogether because the agent must not modify it to fit its own behavior.

The last component of the learning agent is the problem generator. It is responsible for suggesting actions that will lead to new and informative experiences. If the performance element had its way, it would keep doing the actions that are best, given what it knows, but if the agent is willing to explore a little and do some perhaps suboptimal actions in the short run, it might discover much better actions for the long run. The problem generator’s job is to suggest these exploratory actions. This is what scientists do when they carry out experiments. Galileo did not think that dropping rocks from the top of a tower in Pisa was valuable in itself. He was not trying to break the rocks or to modify the brains of unfortunate pedestrians. His aim was to modify his own brain by identifying a better theory of the motion of objects.

Problem generator

The learning element can make changes to any of the “knowledge” components shown in the agent diagrams (Figures 2.9 , 2.11 , 2.13 , and 2.14). The simplest cases involve learning directly from the percept sequence. Observation of pairs of successive states of the environment can allow the agent to learn “What my actions do” and “How the world evolves” in response to its actions. For example, if the automated taxi exerts a certain braking pressure when driving on a wet road, then it will soon find out how much deceleration is actually achieved, and whether it skids off the road. The problem generator might identify certain parts of the model that are in need of improvement and suggest □□ □□experiments, such as trying out the brakes on different road surfaces under different conditions.

Improving the model components of a model-based agent so that they conform better with reality is almost always a good idea, regardless of the external performance standard. (In some cases, it is better from a computational point of view to have a simple but slightly inaccurate model rather than a perfect but fiendishly complex model.) Information from the external standard is needed when trying to learn a reflex component or a utility function.

For example, suppose the taxi-driving agent receives no tips from passengers who have been thoroughly shaken up during the trip. The external performance standard must inform the agent that the loss of tips is a negative contribution to its overall performance; then the agent might be able to learn that violent maneuvers do not contribute to its own utility. In a sense, the performance standard distinguishes part of the incoming percept as a reward (or penalty) that provides direct feedback on the quality of the agent's behavior. Hard-wired performance standards such as pain and hunger in animals can be understood in this way.

Reward

Penalty

More generally, human choices can provide information about human preferences. For example, suppose the taxi does not know that people generally don't like loud noises, and settles on the idea of blowing its horn continuously as a way of ensuring that pedestrians know it's coming. The consequent human behavior—covering ears, using bad language, and possibly cutting the wires to the horn—would provide evidence to the agent with which to update its utility function. This issue is discussed further in Chapter 22 .

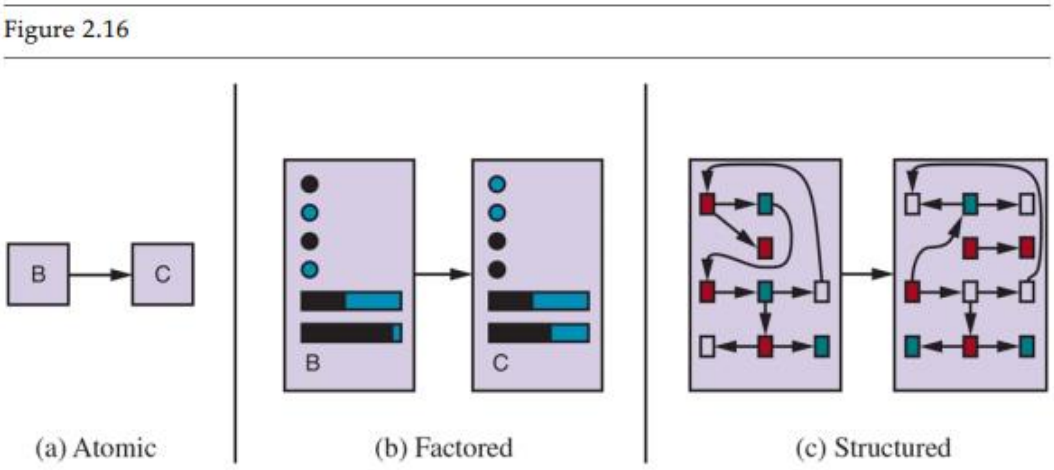
In summary, agents have a variety of components, and those components can be represented in many ways within the agent program, so there appears to be great variety among learning methods. There is, however, a single unifying theme. Learning in intelligent agents can be summarized as a process of modification of each component of the agent to bring the components into closer agreement with the available feedback information, thereby improving the overall performance of the agent.

2.4.7 How the components of agent programs work

We have described agent programs (in very high-level terms) as consisting of various components, whose function it is to answer questions such as: "What is the world like now?" "What action should I do now?" "What do my actions do?" The next question for a student of AI is, "How on Earth do these components work?" It takes about a thousand pages to begin to answer that question properly, but here we want to draw the reader's attention to some basic distinctions among the various ways that the components can represent the environment that the agent inhabits.

Roughly speaking, we can place the representations along an axis of increasing complexity and expressive power—atomic, factored, and structured. To illustrate these ideas, it helps to consider a particular agent component, such as the one that deals with "What my actions do." This component describes the changes that might occur in the environment as the result of taking an action, and Figure 2.16 provides schematic depictions of how those transitions might be represented.

Figure 2.16



Three ways to represent states and the transitions between them. (a) Atomic representation: a state (such as B or C) is a black box with no internal structure; (b) Factored representation: a state consists of a vector of attribute values; values can be Boolean, real-valued, or one of a fixed set of symbols. (c) Structured representation: a state includes objects, each of which may have attributes of its own as well as relationships to other objects.

Three ways to represent states and the transitions between them. (a) Atomic representation: a state (such as B or C) is a black box with no internal structure; (b) Factored representation: a state consists of a vector of attribute values; values can be Boolean, real-valued, or one of a fixed set of symbols. (c) Structured representation: a state includes objects, each of which may have attributes of its own as well as relationships to other objects.

In an atomic representation each state of the world is indivisible—it has no internal structure. Consider the task of finding a driving route from one end of a country to the other via some sequence of cities (we address this problem in Figure 3.1 on page 64). For the purposes of solving this problem, it may suffice to reduce the state of the world to just the name of the city we are in—a single atom of knowledge, a "black box" whose only discernible property is that of being identical to or different from another black box. The standard algorithms underlying search and game-playing (Chapters 3 –5), hidden Markov models (Chapter 14), and Markov decision processes (Chapter 17) all work with atomic representations.

Atomic representation

A factored representation splits up each state into a fixed set of variables or attributes, each of which can have a value. Consider a higher-fidelity description for the same driving problem, where we need to be concerned with more than just atomic location in one city or another; we might need to pay attention to how much gas is in the tank, our current GPS coordinates, whether or not the oil warning light is working, how much money we have for tolls, what station is on the radio, and so on. While two different atomic states have nothing in common—they are just different black boxes—two different factored states can share some attributes (such as being at some particular GPS location) and not others (such as having lots of gas or having no gas); this makes it much easier to work out how to turn one state into another. Many important areas of AI are based on factored representations, including constraint satisfaction algorithms (Chapter 6), propositional logic (Chapter 7), planning (Chapter 11), Bayesian networks (Chapters 12 –16), and various machine learning algorithms.

Factored representation

Variable

Attribute

Value

For many purposes, we need to understand the world as having things in it that are related to each other, not just variables with values. For example, we might notice that a large truck ahead of us is reversing into the driveway of a dairy farm, but a loose cow is blocking the truck's path. A factored representation is unlikely to be pre-equipped with the attribute *TruckAheadBackingIntoDairyFarmDrivewayBlockedByLooseCow* with value true or false. Instead, we would need a structured representation, in which objects such as cows and trucks and their various and varying relationships can be described explicitly (see Figure 2.16(c)). Structured representations underlie relational databases and first-order logic (Chapters 8 , 9 , and 10), first-order probability models (Chapter 15), and much of natural language understanding (Chapters 23 and 24). In fact, much of what humans express in natural language concerns objects and their relationships.

Structured representation

As we mentioned earlier, the axis along which atomic, factored, and structured representations lie is the axis of increasing expressiveness. Roughly speaking, a more expressive representation can capture, at least as concisely, everything a less expressive one can capture, plus some more. Often, the more expressive language is much more concise; for example, the rules of chess can be written in a page or two of a structured-representation language such as first-order logic but require thousands of pages when written in a factored representation language such as propositional logic

and around pages when written in an atomic language such as that of finite-state automata. On the other hand, reasoning and □□□□□□□ 10 38learning become more complex as the expressive power of the representation increases. To gain the benefits of expressive representations while avoiding their drawbacks, intelligent systems for the real world may need to operate at all points along the axis simultaneously.

Expressiveness

Localist representation

Another axis for representation involves the mapping of concepts to locations in physical memory, whether in a computer or in a brain. If there is a one-to-one mapping between concepts and memory locations, we call that a localist representation. On the other hand, if the representation of a concept is spread over many memory locations, and each memory location is employed as part of the representation of multiple different concepts, we call that a distributed representation. Distributed representations are more robust against noise and information loss. With a localist representation, the mapping from concept to memory location is arbitrary, and if a transmission error garbles a few bits, we might confuse Truck with the unrelated concept Truce. But with a distributed representation, you can think of each concept representing a point in multidimensional space, and if you garble a few bits you move to a nearby point in that space, which will have similar meaning.

Distributed representation